IA-64 Porting Methodology

An Application Guide

Table of Contents

Introduction

This document presents a methodology for porting applications from IA-32 to IA-64 architecture. In addition to porting an application to IA-64 architecture, this methodology helps you maintain backward IA-32 compatibility with the same source code using minimal conditional compilation, a technique often referred to as “single source”. The methodology described is the best-known method for systematically porting applications. It draws on the experience gained by Intel software engineers who assist software companies port large applications to IA-64. This document does not provide details or “how-to” for each step in the methodology. References listed at the end of this document address the “how-to”.

Methodology Overview

There are eight basic steps in the porting methodology as illustrated in Figure 1. As shown, many of the steps may be done concurrently.

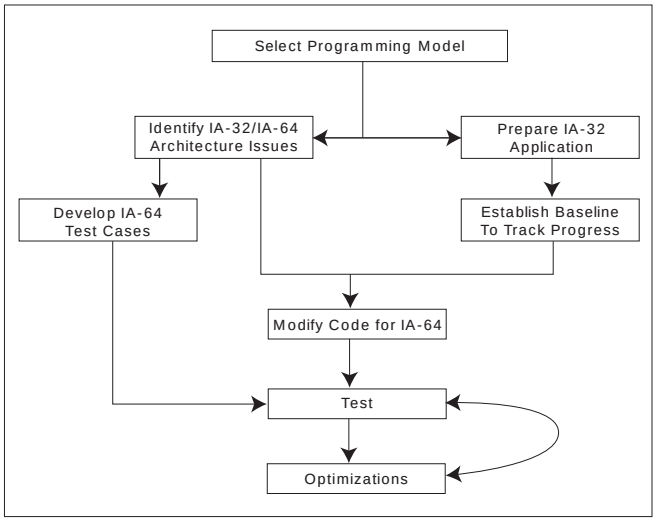

Figure 1: Methodology Overview

- **Select the appropriate programming model.** Although operating system and compiler dependent, there are four general programming models supported. The specific models are discussed in the following section on Programming Models. Since a complete native IA-64 port is the most invasive programming model, this document assumes the native IA-64 programming model is selected.
- **Prepare the IA-32 application.** Preparing the IA-32 application is cleaning up the IA-32 implementation by resolving compiler warnings, removing unused libraries, etc. A number of porting issues can be avoided by polishing the IA-32 application before any IA-64 porting is started.
- **Identify IA-32/IA-64 application issues.** There are a number of issues that require additional engineering and code to handle differences between the IA-32 and IA-64 architectures. For example, source code that reads or writes application specific files might be freed of 32-bit capacity restrictions present in the 32-bit version of the application. In addition, you may need to add or modify code to handle both IA-32 and IA-64 data formats as well as provide automatic detection and conversion between the two.
- **Establish a baseline.** This involves modifying the build environment to support an IA-64 build and start tracking the number of issues (errors and warnings). As you port the code, the number of errors and warnings during compilation/linking can be used to measure your porting progress.
- **Begin code modifications to implement the IA-64 version of the application.** Start by eliminating compilation errors, then linker errors, followed by eliminating compilation warnings, and finally resolving run-time errors.
- **Develop specific test cases.** Once the IA-32/IA-64 application issues have been identified, you can use those issues to develop specific test cases to verify the correct implementation during the testing phase.
- **The final phases are testing and optimization.**

1. Select the Programming Model

A significant number of IA-32 applications exist today. Those same applications run on IA-32 processors that did not exist at the time the code was originally written. Intel® Itanium™ processor architects as well as operating system and compiler engineers recognize the tremendous value of backward capability. Therefore, the architects and engineers have provided the capability to support several programming models. The programming models available range from full native IA-64 programs with the full performance, capacity, and bandwidth potential available from the processor's new IA-64 architecture to unmodified IA-32 programs without the full benefits available from the IA-64 architecture.

Note that the programming models supported for Intel's Itanium processor are operating system and compiler dependent.

■ *Native 64-bit Programming Model*

The native 64-bit programming model gains all of the benefits of moving to a 64-bit platform such as improved performance and supporting larger memory requirements. This also requires the most porting effort from a 32-bit application to a 64-bit application. This programming model is targeted at server and high-end workstation applications that need to take advantage of higher performance, greater bandwidth, and larger data sets.

■ *64-bit API and 32-bit Addressing Programming Model*

It is possible to enjoy the performance benefit of the native IA-64 instruction set while application capacities (and pointers) remain 32-bits in size. Since many applications will not need full 64-bit capacity; this programming model simplifies the IA-64 porting effort. Depending on the operating system, this may be supported via 32-bit (but IA-64 native) libraries or by compiler assistance in safe truncation and extension of pointer values passed to/from the libraries.

■ *32-bit API and 32-bit Addressing Programming Model*

A 32-bit API and 32-bit addressing programming model is a compromise that takes advantage of the native 64-bit architecture and instruction set while minimizing the porting effort. Again, the 32-bit addressing limits the application to a smaller 2GB addressable memory size. Moreover, the 32-bit API may limit performance because it is based on a layer of software that translates the 32-bit API call to a 64-bit API call. The porting effort for this programming model is a full re-build of the application and testing.

■ *Original IA-32 Programming Model*

Intel's 64-bit architecture as implemented in the Intel Itanium processor is backward compatible with Intel's x86 family of processors to up the Intel® Pentium® III processor. An IA-32 binary application compatible with an Intel Pentium III processor or earlier may execute unmodified on IA-64 platforms assuming the operating system enables this feature. Microsoft Windows* 2000 with IA-64-bit hardware supports most existing IA-32 binary applications without modification but without the performance and large data sets available for 64-bit applications. The level of effort for porting an existing IA-32 application should be limited to ensuring the application is compatible with the new operating system.

2. Pre-porting Preparation

Numerous porting issues arise while porting an application to the IA-64 platform. The porting issues can be divided into three broad categories:

- Application Dependencies
- IA-32 implementation issues not supported in IA-64 environment
- IA-64 implementation issues not supported in IA-32 environment

You are more likely to speed up the overall porting process by addressing these issues in a logical order while minimizing the impact across an application. Therefore, the next step is to make sure the application is prepared for porting before you begin.

The goal of this step is to eliminate IA-64 porting issues while still in the IA-32 environment. Remaining in the IA-32 environment reduces the number of issues you need to address and they can usually be addressed independently from one another. This also allows you to test the modifications without impacts from IA-64 specific issues.

Identify Application Dependencies

Identify all of your application dependencies and begin the process to make sure they are going to be compatible with the target IA-64 environment. These include third-party software components such as libraries, DLLs, in-process/out-of-process server applications, security dongles, as well as installation programs, testing utilities, and tools used in your build environment.

Dependencies that are external to your organization and not compatible with the target IA-64 environment may turn out to take the longest time to acquire. In addition to the technical porting effort, business issues need to be resolved. It is best to engage the responsible third parties early and establish your plan of record at the beginning. The results of these early engagements may have a direct impact on when or whether you will port the application at all.

If a dependency is not ported to the target IA-64 environment, there are many options around the obstacle. You may:

- contractually engage the third party to meet your needs
- Replace the dependency with an alternative existing third-party component compatible with the IA-64 platform
- Build an IA-32/IA-64 interface depending on the current IA-32 implementation of the dependency
- Build your own implementation

Remove unused code/libraries

It is not unusual for source code or build environments to include more code or libraries than the final application or dynamic-linked library requires. This can happen when an application is initially created or over time as the code is modified. For example, Microsoft Visual Studio* project wizard may add support for functionality based on the programmer's

selections that are never used. Consequently, headers and libraries will be added to the project. Alternatively, code is modified over time without removing the all the source, headers, or libraries affected.

Remove unused code and libraries from your source and build environments. Without removing the unused code, you may find yourself porting code that is never executed. Or, you may track dependencies that are not necessary for the application.

Eliminate Self-modifying Code

Intel's 64-bit instruction set is incompatible with the IA-32 instruction set. If the application being ported has self-modifying code, then assess the importance of this code to the application. If the functionality which requires the self-modifying code can be modified to not require self-modifying code, then eliminate this code. If not, wait until you are porting the code for the IA-64 environment. At which time, you will need to re-write/re-engineer the code for IA-64 and use conditional compilation to select the appropriate IA-32/IA-64 code for the target platform.

Eliminate Compiler Warnings

It is not unusual for commercially released software to have compilation warnings during the build process. Over time, many software engineers come to accept the warnings without thought. They recognize the warning and so far, it has not resulted in anomalies in the execution of the application. The problem with accepting the warnings and beginning the porting to IA-64 is twofold. First, compiling the application with an IA-64 compiler is likely to generate many new serious warnings. The new warnings are likely to be type-size mismatch issues that that need to be resolved. It will become more difficult to distinguish between the "OK to ignore" warnings and the "serious application impacting" warnings. Second, the existing IA-32 warnings may have serious impact in an IA-64 application. By understanding and resolving the IA-32 warnings with forethought for implications for IA-64, you can avoid bugs further in the porting process.

Eliminate Module Definition Files

Microsoft Windows uses module definition files (DEF) to identify exported symbols from Dynamically Linked Libraries (DLLs). This is more typical of legacy software before IA-32 Windows applications but is still used today. If you build DLLs or use DLLs, try to eliminate module definition files by using the appropriate dllimport / dllexport syntax. Modifying the source to use this syntax is straightforward. You can use the existing IA-32 DEF file to help identify the needed exported symbols. You will need to use conditional compilation in the header files so that dllimport modifies the symbol declarations when used in your executable but uses dllexport when used in the DLL.

3. Baseline IA-64 Application

Before porting an application to IA-64, the application should be well understood. That is, the build environment, the source code, test cases, test data and test results should be frozen. The software engineers porting the application should know the expected behavior for the IA-32 application. The pre-porting preparation should establish the initial IA-64 porting baseline.

Build Environment Established

At least initially, most IA-64 porting environments will be cross-platform based. Cross-platform means that the application will be compiled and built on one platform, mostly likely an IA-32 platform, and the resulting application runs on the target IA-64 platform. Both Microsoft and Intel have developed cross-platform compilers to support IA-64 development for Microsoft Windows 2000. Moreover, existing scripting tools that do not generate machine instructions, such as perl, python, tlk, etc., are available. Fortunately, internally developed tools should also continue to work in the build environment if they do not generate or tweak machine instructions.

Modify Makefile, Project, and Scripts

Modify makefiles, Microsoft Visual C/C++* projects, and scripts to support IA-64 porting. The standard includes and libraries have changed for IA-64. Refer to your specific IA-64 compiler/linker documentation for appropriate settings. Using the Microsoft or Intel compiler for Windows 2000, you will need to modify the default executable, include, and library paths under the options menu as well as the compiler and linker settings for the specific project.

Tracking Progress

Every software project needs a schedule and a method to track progress against that schedule. Without historical data or experience, developing the schedule can be a challenge especially when you aren't sure how much work there is to be completed. Fortunately, the compiler can help establish an initial baseline and track progress that can be used to project target milestones.

During the initial porting process, track the number of compilation errors, compilation warnings, and linked modules against time. You can use this data to track your progress against the schedule.

4. Identify Known IA-32/IA-64 Application Issues

For some applications, completing all the pre-porting preparation will resolve all of the IA-32/IA-64 application issues and, after recompilation, only testing/optimization tasks remain. For other applications, IA-32/IA-64 application issues can not be solved with a single source-code base. Additional software engineering and code development will be required along with conditional compilation.

First, all assembly code must be replaced with C/C++ code or augmented with IA-64 assembly code and use appropriate conditional compilation.

Since the stack is completely different for the IA-64 implementation, any code dealing directly with the stack needs to be augmented with appropriate IA-64 code even if the current IA-32 code is using appropriate operating system APIs. Although the APIs may be similar, they will be sufficiently different to require re-coding.

Attention should be applied to external interfaces including application specific file formats. File formats may not support larger data sets particularly if indices are part of the format specification. Other external interfaces are also at risk. Review inter-process communication between platforms especially if the application will support both IA-32 and IA-64 platforms.

5. Develop IA-64 Test Plan

Testing is a significant portion of the software development process and porting to IA-64 is no exception. In reviewing the testing environment early in the dependency evaluation, you may have identified issues with being able to test your application in an IA-64 environment. In addition, new test cases to validate the symbolal and performance capabilities need to be developed. By the time you are ready for unit, module, or application-level testing, your test utilities automated regression test suites, and test cases should be available.

Review/Update IA-32 Test Cases

Fortunately, you should be able to re-use a significant portion of the test cases developed for testing your application on IA-32. However, each IA-32 test case should be reviewed for potential changes because of porting to IA-64. Particularly, review input and expected results. You will need to expand the input/output boundary conditions. The impacts may be subtle. For example, a view that displays a count field may not be large enough now that the data set can be larger than 232.

New Test Cases to Expand the Problem Space

The significant benefit to IA-64 is the ability to increase the problem space that you can solve for your end users. Add test cases to expand the amount of memory required by your application. Update performance benchmarks to identify increased computational workloads. These tests are great for validating application symbolality and stability but more importantly, they can generate great marketing material.

Develop New IA-64 Test Cases

Start with the IA-32/IA-64 application issues previously identified. With new or modified code comes the requirement for new unit, module, and application level test cases. Also, focus on areas that have been modified using conditional compilation.

6. “Code Cleaning”

Application Programming Interfaces (APIs) and type definitions for many operating systems targeted at IA-64 have been developed allowing a single source-code base with minimal conditional compilation to build either IA-32 or IA-64 versions of an application. For example, Microsoft created the 64-bit version of the Windows operating system with this goal in mind for Win32*- and Win64*-based applications. This support is available in Microsoft Platform SDK for the Windows 2000 operating system. The process for achieving the single source compatibility is often referred to as “Code Cleaning”.

Eliminate Compilation Errors

The first step to code cleaning is to eliminate compilation errors. This is straightforward since the IA-64 compiler will readily identify the source of the compilation errors. Usually, the causes of compilation errors at this stage are the result of API changes. Refer to the specific operating system documentation for which APIs have changed. You may also have a large number of errors because of IA-32 assembly code has not been conditionally compiled.

Now is the time to replace/rewrite the assembly code with appropriate IA-64 code. Less obvious errors are the result of ambiguous symbol or method calls that can not be implicitly resolved because a new IA-64 native type has been introduced.

Code Cleaning Link-time Errors

The second step to code cleaning is to eliminate/resolve link time errors. Most link time errors are the result of unavailable IA-64 libraries, linking an IA-32 library version, or using module definition files built for IA-32. The goal is to have a fully compiled/linked executable and associated dynamically linked components.

Code Cleaning Warnings

The next step is to resolve code clean warnings. These type-size mismatches result in truncation or expansion of a data type. Particularly, they result when a 64-bit pointer is assigned to a smaller size variable or vice versa. Type-size mismatches are likely to be the source of most run-time errors. Depending on the source code, you are likely to get a large quantity of warnings, which is why it is best to eliminate all IA-32 compilation warnings in the pre-porting phase.

This step requires a knowledgeable engineer familiar with the application and its architecture. It is easy to eliminate these warnings using the cast operator. However, this does not resolve the problem. The problem may be masked and, thus, more difficult to find later in the test/debugging phase. The cast operator should be used sparingly and only when the side effect is known to be benign.

Eliminate Run-time Errors

The most difficult errors to eliminate are run-time errors. This step is part of the test, debug, and correct cycle. Be particularly wary of code that has specific application knowledge of data types, structures, or buffers and importantly their size. With the increased size of a pointer, there is potential for run-time errors due to memory overrun or unaligned accesses.

For example, in Microsoft Windows or X-Windows, the programmer can add extra data to a class, window structure, or callbacks. In the case of Microsoft Windows, the cbWndExtra or cbClsExtra structures specify the number of extra bytes appended to the data structure. If pointers are stored within the extra data, then the number of bytes needs to be increased for IA-64. This is an example where the compiler is unlikely to generate a warning or error and will most likely result in a run-time error.

Unaligned data accesses are another source of run-time errors and degraded run-time performance. The Itanium processor will fault on some unaligned data accesses. Not all operating systems will handle the fault. Microsoft Windows 2000 operating system will handle the fault but with a significant performance penalty. If this is not a run-time error for your operating system, it will be an issue to be resolved during optimization.

There are many potential sources of run-time errors. Remember that this is a new platform with a new operating system, new compiler/linker, and a host of new libraries with the potential to have bugs in any or all layers. Therefore, errors from these components should not be ruled out. However, assume that a run-time error is the result an application coding problem initially unless you can determine otherwise.

Tools for Code Clean

The MigraTEC* Migration Workbench is a tool that can help with the actual code modification phase of the porting methodology. It focuses on analyzing the source with respect to porting issues. MigraTEC currently supports porting applications from IA-32 to IA-64 for both Windows 2000 and UNIX*. Refer to www.migratec.com for more details.

Typical "Code Cleaning" Issues

- Invalid cast between pointers and integer and/or long variables
- Invalid printf specifier
- Usage of existing APIs with pointer and/or long as parameters or return values
- Usage of undocumented or reserved bit fields
- Hard-coded values for sizes of data types
- Hard-coded values for bit-shift values
- Hard-coded constants in memory allocation functions
- Unguarded conditional compilation, such as ifdefs, from defaulting to unwanted code generation
- Accessing data structure members via constant offsets
- Algorithms that make use of special bits in pointer arithmetic assuming fixed size
- In-line assembly code
- Self-modifying code
- Appropriately modify portions of code storing/restoring on-disk structures
- Portions of code utilizing data-packing
- For more details, refer to ftp://download.intel.com/design/servers/softdev/CodeClean_r02.pdf

7. Testing

In order to deliver a stable and reliable product on the IA-64 platform, extensive testing is required. You should expect the amount of effort required to test both an IA-32 and IA-64 application will more than double what is required for testing just the IA-32 implementation.

There are likely to be changes to common source code across platforms as well as new code specific to one or more platforms, which need to be tested. To ensure the existing functionality works as expected, use existing IA-32 test suites and test data to test the application on existing IA-32 platforms currently supported. Add IA-64 test cases and test data to ensure that the IA-32 platform behaves as expected even if it means handling fatal errors gracefully. After re-compiling/linking the source for IA-64 testing, use the same IA-32 test suites and test data to ensure matching functionality with the IA-32 implementation. Next, using new IA-64 test cases and test data, test the IA-64 application to ensure correct functionality.

8. Optimization

The final step to porting is the optimization step. Depending on the application, this step may be a very critical activity that unlocks the full potential of the IA-64 architecture. Optimization is the step that focuses on run-time performance and resource utilization. Using Intel Vtune™ Performance Analyzer software can help identify a variety of bottlenecks and sources of performance penalties. You may find that you can limit data expansion due to pointer sizes by using base addressing and indices. Additionally, you may eliminate unaligned data accesses to boost run-time performance.

Summary

For large complex applications, porting to an IA-64 platform is straightforward if not easy. The key is to sub-divide the problem into separate steps. Try not to mix issues between the steps. Start with a very clean and up-to-date version of your IA-32 application. By resolving IA-32 issues early, it will be easier to see the IA-64 issues and not confuse them. Identify areas that you know you will likely have an IA-64 issue and begin working on them next. These areas are more likely the areas that will require some re-coding or re-engineering to resolve the issue. These same areas need additional test cases. They are also likely to be required early in the porting process to resolve compilation/linking issues. Fix compilation/linking errors before focusing on the warnings. Next, resolve warnings and then run-time issues. Finally, be sure to test each platform supported. Also, don’t forget that optimization is a critical step in porting to achieve the highest performance and minimize overall resource utilization.

References

The following documents and Web sites provide background and supporting information for understanding the topics in this document.

"Preparing Code for the IA-64 Architecture (Code Clean)", Intel Corporation, Order # 245435-001.

Intel® Code Clean Enables Software in Both IA-32 and IA-64 Worlds
http://developer.intel.com/update/archive/issue23/stories/top1.htm

Microsoft Getting Ready for 64-bit Windows
http://msdn.microsoft.com/library/psdk/buildapp/64bitwin_410z.htm

Linux* on IA-64
http://www.linuxia64.org

Project Monterey
http://www.ibm.com/servers/monterey/

SCO UnixWare* 64 Bit Porting Guide (Monterey)
http://www.sco.com/developer/64bit.htm

HP* IA-64 Transition Kit
http://www.software.hp.com/products/IA64/index.html

Sun* Solaris* 7 with 64 bits
http://www.sun.com/developers/64bits/

Novell* Modesto
http://www.novell.com/whitepapers/iw/modesto.html

MigraTEC
http://www.migratec.com/